



Índice

1. algorithm	2
2. Estructuras	3
2.1. RMQ (static)	3
2.2. RMQ (dynamic)	3
2.3. RMQ (lazy)	3
2.4. Fenwick Tree	4
2.5. Union Find	4
2.6. Disjoint Intervals	4
2.7. RMQ (2D)	4
2.8. Big Int	5
2.9. Modnum	6
2.10. Treap	7
2.11. Gain-Cost Set	8
3. Algos	8
3.1. Longest Increasing Subsequence	8
3.2. Manacher	8
4. Strings	9
4.1. KMP	9
4.2. Trie	9
4.3. Suffix Array (corto, $n \log 2n$)	9
4.4. Suffix Array (largo, $n \log n$)	9
4.5. String Matching With Suffix Array	10
4.6. LCP (Longest Common Prefix)	10

4.7. Corasick	10
5. Geometría	11
5.1. Punto	11
5.2. Line	11
5.3. Segment	12
5.4. Rectangle	12
5.5. Polygon Area	12
5.6. Circle	12
5.7. Point in Poly	12
5.8. Convex Check CHECK	13
5.9. Convex Hull	13
5.10. Cut Polygon	13
5.11. Bresenham	13
5.12. Rotate Matrix	13
6. Math	14
6.1. Identidades	14
6.2. Combinatorio	14
6.3. Exp. de Numeros Mod.	14
6.4. Exp. de Matrices y Fibonacci en $\log(n)$	14
6.5. Teorema Chino del Resto	14
6.6. Funciones de primos	14
6.7. Phollard's Rho (rolando)	15
6.8. Criba	16
6.9. Factorizacion	16
6.10. GCD	16
6.11. LCM	16
6.12. Inversos	16
6.13. Simpson	16
6.14. Fraction	17
6.15. Polinomio	17
7. Grafos	18
7.1. Dijkstra	18
7.2. Bellman-Ford	18
7.3. Floyd-Warshall	18
7.4. Kruskal	18
7.5. Prim	19
7.6. 2-SAT + Tarjan SCC	19
7.7. Articulation Points	19
7.8. Comp. Biconexas y Puentes	20
7.9. LCA + Climb	20

7.10. Heavy Light Decomposition	21
8. Network Flow	21
8.1. Dinic	21
8.2. Konig	22
8.3. Edmonds Karp's	22
8.4. Push-Relabel	23
9. Ayudamemoria	23

1. algorithm

#include <algorithm> #include <numeric>

Algo	Params	Funcion
sort, stable_sort	f, l	ordena el intervalo
nth_element	f, nth, l	<i>void</i> ordena el n-esimo, y particiona el resto
fill, fill_n	f, l / n, elem	<i>void</i> llena [f, l) o [f, f+n) con elem
lower_bound, upper_bound	f, l, elem	<i>it</i> al primer / ultimo donde se puede insertar elem para que quede ordenada
binary_search	f, l, elem	<i>bool</i> esta elem en [f, l)
copy	f, l, resul	hace resul+i=f+i $\forall i$
find, find_if, find_first_of	f, l, elem / pred / f2, l2	<i>it</i> encuentra i $\in [f, l)$ tq. i=elem, pred(i), $i \in [f2, l2)$
count, count_if	f, l, elem/pred	cuenta elem, pred(i)
search	f, l, f2, l2	busca [f2, l2) $\in [f, l)$
replace, replace_if	f, l, old / pred, new	cambia old / pred(i) por new
reverse	f, l	da vuelta
partition, stable_partition	f, l, pred	pred(i) ad, !pred(i) atras
min_element, max_element	f, l, [comp]	<i>it</i> min, max de [f, l]
lexicographical_compare	f1, l1, f2, l2	<i>bool</i> con [f1, l1] _i [f2, l2]
next/prev_permutation	f, l	deja en [f, l) la perm sig, ant
set_intersection, set_difference, set_union, set_symmetric_difference,	f1, l1, f2, l2, res	[res, ...) la op. de conj
push_heap, pop_heap, make_heap	f, l, e / e /	mete/saca e en heap [f, l), hace un heap de [f, l)
is_heap	f, l	<i>bool</i> es [f, l) un heap
accumulate	f, l, i, [op]	$T = \sum / \text{oper de } [f, l)$
inner_product	f1, l1, f2, i	$T = i + [f1, l1) \cdot [f2, \dots)$
partial_sum	f, l, r, [op]	$r+i = \sum / \text{oper de } [f, f+i) \forall i \in [f, l)$
__builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
__builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
__builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
__builtin_popcount	unsigned int	Cant. de 1's en x.
__builtin_parity	unsigned int	1 si x es par, 0 si es impar.
__builtin_XXXXXXll	unsigned ll	= pero para long long's.

2. Estructuras

2.1. RMQ (static)

Dado un arreglo y una operación asociativa *idempotente*, `get(i, j)` opera sobre el rango `[i, j]`. Restricción: $LVL \geq \text{ceil}(\log n)$; Usar `[]` para llenar arreglo y luego `build()`.

```

1 struct RMQ{
2     #define LVL 10
3     tipo vec[LVL][1<<(LVL+1)];
4     tipo &operator[] (int p){return vec[0][p];}
5     tipo get(int i, int j) { //intervalo [i,j]
6         int p = 31-__builtin_clz(j-i);
7         return min(vec[p][i], vec[p][j-(1<<p)]);
8     }
9     void build(int n) { //O(nlogn)
10        int mp = 31-__builtin_clz(n);
11        forn(p, mp) forn(x, n-(1<<p))
12            vec[p+1][x] = min(vec[p][x], vec[p][x+(1<<p)]);
13    }
14 };

```

2.2. RMQ (dynamic)

```

1 //Dado un arreglo y una operacion asociativa con neutro, get(i, j) opera
   sobre el rango [i, j].
2 #define MAXN 100000
3 #define operacion(x, y) max(x, y)
4 const int neutro=0;
5 struct RMQ{
6     int sz;
7     tipo t[4*MAXN];
8     tipo &operator[] (int p){return t[sz+p];}
9     void init(int n){ //O(nlgn)
10        sz = 1 << (32-__builtin_clz(n));
11        forr(i, sz, 2*sz) t[i]=neutro;
12    }
13    void updall(){ //O(n)
14        dforn(i, sz) t[i]=operacion(t[2*i], t[2*i+1]);}
15    tipo get(int i, int j){return get(i,j,1,0,sz);}
16    tipo get(int i, int j, int n, int a, int b){ //O(lgn)
17        if(j<=a || i>=b) return neutro;
18        if(i<=a && b<=j) return t[n];

```

```

19        int c=(a+b)/2;
20        return operacion(get(i, j, 2*n, a, c), get(i, j, 2*n+1, c, b));
21    }
22    void set(int p, tipo val){ //O(lgn)
23        for(p+=sz; p>0 && t[p]!=val;){
24            t[p]=val;
25            p/=2;
26            val=operacion(t[p*2], t[p*2+1]);
27        }
28    }
29 }rmq;
30 //Usage:
31 cin >> n; rmq.init(n); forn(i, n) cin >> rmq[i]; rmq.updall();

```

2.3. RMQ (lazy)

```

1 //Dado un arreglo y una operacion asociativa con neutro, get(i, j) opera
   sobre el rango [i, j].
2 typedef int Elem; //Elem de los elementos del arreglo
3 typedef int Alt; //Elem de la alteracion
4 #define operacion(x,y) x+y
5 const Elem neutro=0; const Alt neutro2=0;
6 #define MAXN 100000
7 struct RMQ{
8     int sz;
9     Elem t[4*MAXN];
10    Alt dirty[4*MAXN]; //las alteraciones pueden ser de distinto Elem
11    Elem &operator[] (int p){return t[sz+p];}
12    void init(int n){ //O(nlgn)
13        sz = 1 << (32-__builtin_clz(n));
14        forr(i, sz, 2*sz) t[i]=neutro;
15        forn(i, 2*sz) dirty[i]=neutro2;
16    }
17    void push(int n, int a, int b){ //propaga el dirty a sus hijos
18        if(dirty[n]!=0){
19            t[n]+=dirty[n]*(b-a); //altera el nodo
20            if(n<sz){
21                dirty[2*n]+=dirty[n];
22                dirty[2*n+1]+=dirty[n];
23            }
24            dirty[n]=0;
25        }
26    }

```

```

27 Elem get(int i, int j, int n, int a, int b){//0(lgn)
28     if(j<=a || i>=b) return neutro;
29     push(n, a, b);//corrige el valor antes de usarlo
30     if(i<=a && b<=j) return t[n];
31     int c=(a+b)/2;
32     return operacion(get(i, j, 2*n, a, c), get(i, j, 2*n+1, c, b));
33 }
34 Elem get(int i, int j){return get(i,j,1,0,sz);}
35 //altera los valores en [i, j) con una alteracion de val
36 void alterar(Alt val, int i, int j, int n, int a, int b){//0(lgn)
37     push(n, a, b);
38     if(j<=a || i>=b) return;
39     if(i<=a && b<=j){
40         dirty[n]+=val;
41         push(n, a, b);
42         return;
43     }
44     int c=(a+b)/2;
45     alterar(val, i, j, 2*n, a, c), alterar(val, i, j, 2*n+1, c, b);
46     t[n]=operacion(t[2*n], t[2*n+1]);//por esto es el push de arriba
47 }
48 void alterar(Alt val, int i, int j){alterar(val,i,j,1,0,sz);}
49 }rmq;

```

2.4. Fenwick Tree

```

1 //For 2D threat each column as a Fenwick tree, by adding a nested for in
  each operation
2 struct Fenwick{
3     static const int sz=1000001;
4     tipo t[sz];
5     void adjust(int p, tipo v){//valid with p in [1, sz), 0(lgn)
6         for(; p<sz; p+=(p&-p)) t[p]+=v; }
7     tipo sum(int p){//cumulative sum in [1, p], 0(lgn)
8         tipo s=0;
9         for(; p; p--(p&-p)) s+=t[p];
10        return s;
11    }
12    tipo sum(int a, int b){return sum(b)-sum(a-1);}
13    //get largest value with cumulative sum less than or equal to x;
14    //for smallest, pass x-1 and add 1 to result
15    int getind(tipo x) {//0(lgn)
16        int idx = 0, mask = N;

```

```

17        while(mask && idx < N) {
18            int t = idx + mask;
19            if(x >= tree[t])
20                idx = t, x -= tree[t];
21            mask >>= 1;
22        }
23        return idx;
24    }
25 };

```

2.5. Union Find

```

1 struct UnionFind{
2     vector<int> f;//the array contains the parent of each node
3     void init(int n){f.clear(); f.insert(f.begin(), n, -1);}
4     int comp(int x){return (f[x]==-1?x:f[x]=comp(f[x]));} //0(1)
5     bool join(int i, int j) {
6         bool con=comp(i)==comp(j);
7         if(!con) f[comp(i)] = comp(j);
8         return con;
9     }
10 };

```

2.6. Disjoint Intervals

```

1 bool operator< (const ii &a, const ii &b) {return a.fst<b.fst;}
2 //Stores intervals as [first, second]
3 //in case of a collision it joins them in a single interval
4 struct disjoint_intervals {
5     set<ii> segs;
6     void insert(ii v) {//0(lgn)
7         if(v.snd-v.fst==0.) return;//0J0
8         set<ii>::iterator it,at;
9         at = it = segs.lower_bound(v);
10        if (at!=segs.begin() && (--at)->snd >= v.fst)
11            v.fst = at->fst, --it;
12        for(; it!=segs.end() && it->fst <= v.snd; segs.erase(it++))
13            v.snd=max(v.snd, it->snd);
14        segs.insert(v);
15    }
16 };

```

2.7. RMQ (2D)

```

1 struct RMQ2D{
2     static const int sz=1024;
3     RMQ t[sz];
4     RMQ &operator[] (int p){return t[sz/2+p];}
5     void build(int n, int m){//0(nm)
6         forr(y, sz/2, sz/2+m)
7             t[y].build(m);
8         forr(y, sz/2+m, sz)
9             forn(x, sz)
10                t[y].t[x]=0;
11         dforr(y, sz/2)
12             forn(x, sz)
13                t[y].t[x]=max(t[y*2].t[x], t[y*2+1].t[x]);
14     }
15     void set(int x, int y, tipo v){//0(lgm.lgn)
16         y+=sz/2;
17         t[y].set(x, v);
18         while(y/=2)
19             t[y].set(x, max(t[y*2].t[x], t[y*2+1].t[x]));
20     }
21     //0(lgm.lgn)
22     int get(int x1, int y1, int x2, int y2, int n=1, int a=0, int b=sz/2){
23         if(y2<=a || y1>=b) return 0;
24         if(y1<=a && b<=y2) return t[n].get(x1, x2);
25         int c=(a+b)/2;
26         return max(get(x1, y1, x2, y2, 2*n, a, c),
27                    get(x1, y1, x2, y2, 2*n+1, c, b));
28     }
29 };
30 //Example to initialize a grid of M rows and N columns:
31 RMQ2D rmq;
32 forn(i, M)
33     forn(j, N)
34         cin >> rmq[i][j];
35 rmq.build(N, M);

```

2.8. Big Int

```

1 #define BASEXP 6
2 #define BASE 1000000
3 #define LMAX 1000
4 struct bint{
5     int l;

```

```

6     ll n[LMAX];
7     bint(ll x=0){
8         l=0;
9         forn(i, LMAX){
10             n[i]=x%BASE;
11             x/=BASE;
12             l+=!!x||!i;
13         }
14     }
15     bint(string x){
16         l=(x.size()-1)/BASEXP+1;
17         fill(n, n+LMAX, 0);
18         ll r=1;
19         forn(i, sz(x)){
20             n[i / BASEXP] += r * (x[x.size()-1-i]-'0');
21             r*=10; if(r==BASE)r=1;
22         }
23     }
24     void out(){
25         cout << n[l-1];
26         dforr(i, l-1) printf("%6.6llu", n[i]);//6=BASEXP!
27     }
28     void invar(){
29         fill(n+1, n+LMAX, 0);
30         while(l>1 && !n[l-1]) l--;
31     }
32 };
33 bint operator+(const bint&a, const bint&b){
34     bint c;
35     c.l = max(a.l, b.l);
36     ll q = 0;
37     forn(i, c.l) q += a.n[i]+b.n[i], c.n[i]=q %BASE, q/=BASE;
38     if(q) c.n[c.l++] = q;
39     c.invar();
40     return c;
41 }
42 pair<bint, bool> lresta(const bint& a, const bint& b) // c = a - b
43 {
44     bint c;
45     c.l = max(a.l, b.l);
46     ll q = 0;
47     forn(i, c.l) q += a.n[i]-b.n[i], c.n[i]=(q+BASE) %BASE, q=(q+BASE)/
        BASE-1;

```

```

48     c.invar();
49     return make_pair(c, !q);
50 }
51 bint& operator-= (bint& a, const bint& b){return a=lresta(a, b).first;}
52 bint operator- (const bint&a, const bint&b){return lresta(a, b).first;}
53 bool operator< (const bint&a, const bint&b){return !lresta(a, b).second;}
54 bool operator<= (const bint&a, const bint&b){return lresta(b, a).second;}
55 bool operator==(const bint&a, const bint&b){return a <= b && b <= a;}
56 bint operator*(const bint&a, ll b){
57     bint c;
58     ll q = 0;
59     forn(i, a.l) q += a.n[i]*b, c.n[i] = q %BASE, q/=BASE;
60     c.l = a.l;
61     while(q) c.n[c.l++] = q %BASE, q/=BASE;
62     c.invar();
63     return c;
64 }
65 bint operator*(const bint&a, const bint&b){
66     bint c;
67     c.l = a.l+b.l;
68     fill(c.n, c.n+b.l, 0);
69     forn(i, a.l){
70         ll q = 0;
71         forn(j, b.l) q += a.n[i]*b.n[j]+c.n[i+j], c.n[i+j] = q %BASE, q
            /=BASE;
72         c.n[i+b.l] = q;
73     }
74     c.invar();
75     return c;
76 }
77 pair<bint, ll> ldiv(const bint& a, ll b){// c = a / b ; rm = a % b
78     bint c;
79     ll rm = 0;
80     dforn(i, a.l){
81         rm = rm * BASE + a.n[i];
82         c.n[i] = rm / b;
83         rm %= b;
84     }
85     c.l = a.l;
86     c.invar();
87     return make_pair(c, rm);

```

```

88 }
89 bint operator/(const bint&a, ll b){return ldiv(a, b).first;}
90 ll operator%(const bint&a, ll b){return ldiv(a, b).second;}
91 pair<bint, bint> ldiv(const bint& a, const bint& b){
92     bint c;
93     bint rm = 0;
94     dforn(i, a.l){
95         if (rm.l==1 && !rm.n[0])
96             rm.n[0] = a.n[i];
97         else{
98             dforn(j, rm.l) rm.n[j+1] = rm.n[j];
99             rm.n[0] = a.n[i];
100            rm.l++;
101        }
102        ll q = rm.n[b.l] * BASE + rm.n[b.l-1];
103        ll u = q / (b.n[b.l-1] + 1);
104        ll v = q / b.n[b.l-1] + 1;
105        while (u < v-1){
106            ll m = (u+v)/2;
107            if (b*m <= rm) u = m;
108            else v = m;
109        }
110        c.n[i]=u;
111        rm-=b*u;
112    }
113    c.l=a.l;
114    c.invar();
115    return make_pair(c, rm);
116 }
117 bint operator/(const bint&a, const bint&b){return ldiv(a, b).first;}
118 bint operator%(const bint&a, const bint&b){return ldiv(a, b).second;}

```

2.9. Modnum

```

1 struct mnum{
2     static const tipo mod=12582917;
3     tipo v;
4     mnum(tipo v=0): v(v%mod) {}
5     mnum operator+(mnum b){return v+b.v;}
6     mnum operator-(mnum b){return v>=b.v? v-b.v : mod-b.v+v;}
7     mnum operator*(mnum b){return v*b.v;}
8     mnum operator^(int n){
9         if(!n) return 1;

```

```

10     return n%2? (*this)^(n/2)*(*this) : (*this)^(n/2);}
11 };

```

2.10. Treap

```

1  typedef int Key;
2
3  typedef struct node *pnode;
4  struct node{
5      Key key;
6      int prior, size;
7      pnode l,r;
8      node(Key key=0, int prior=0): key(key), prior(prior), size(1), l(0),
          r(0) {}
9  };
10 struct treap {
11     pnode root;
12     treap(): root(0) {}
13     int size(pnode p) { return p ? p->size : 0; }
14     int size() { return size(root); }
15     void push(pnode p) {
16         // modificar y propagar el dirty a los hijos aca(para lazy)
17     }
18     // Update function and size from children's values
19     void pull(pnode p) { //recalcular valor del nodo aca (para rmq)
20         p->size = 1 + size(p->l) + size(p->r);
21     }
22     pnode merge(pnode l, pnode r) {
23         if (!l || !r) return l ? l : r;
24         push(l), push(r);
25         pnode t;
26         if (l->prior < r->prior) l->r=merge(l->r, r), t = l;
27         else r->l=merge(l, r->l), t = r;
28         pull(t);
29         return t;
30     } //opcional:
31     void merge(treap t) {root = merge(root, t.root), t.root=0;}
32     //*****KEY OPERATIONS*****//
33     void splitKey(pnode t, Key key, pnode &l, pnode &r) {
34         if (!t) return void(l = r = 0);
35         push(t);
36         if (key <= t->key) splitKey(t->l, key, l, t->l), r = t;
37         else splitKey(t->r, key, t->r, r), l = t;

```

```

38         pull(t);
39     }
40     void insertKey(Key key) {
41         pnode elem = new node(key, rand());
42         pnode t1, t2; splitKey(root, key, t1, t2);
43         t1=merge(t1,elem);
44         root=merge(t1,t2);
45     }
46     void eraseKeys(Key key1, Key key2) {
47         pnode t1,t2,t3;
48         splitKey(root,key1,t1,t2);
49         splitKey(t2,key2, t2, t3);
50         root=merge(t1,t3);
51     }
52     void eraseKey(pnode &t, Key key) {
53         if (!t) return;
54         push(t);
55         if (key == t->key) t=merge(t->l, t->r);
56         else if (key < t->key) eraseKey(t->l, key);
57         else eraseKey(t->r, key);
58         pull(t);
59     }
60     void eraseKey(Key key) {eraseKey(root, key);}
61     pnode findKey(pnode t, Key key) {
62         if (!t) return 0;
63         if (key == t->key) return t;
64         if (key < t->key) return findKey(t->l, key);
65         return findKey(t->r, key);
66     }
67     pnode findKey(Key key) { return findKey(root, key); }
68     //*****POS OPERATIONS*****// No mezclar con las funciones Key
69     //(No funciona con pos:)
70     void splitSize(pnode t, int sz, pnode &l, pnode &r) {
71         if (!t) return void(l = r = 0);
72         push(t);
73         if (sz <= size(t->l)) splitSize(t->l, sz, l, t->l), r = t;
74         else splitSize(t->r, sz - 1 - size(t->l), t->r, r), l = t;
75         pull(t);
76     }
77     void insertPos(int pos, Key key) {
78         pnode elem = new node(key, rand());
79         pnode t1,t2; splitSize(root, pos, t1, t2);
80         t1=merge(t1,elem);

```

```

81     root=merge(t1,t2);
82 }
83 void erasePos(int pos1, int pos2=-1) {
84     if(pos2==-1) pos2=pos1+1;
85     pnode t1,t2,t3;
86     splitSize(root,pos1,t1,t2);
87     splitSize(t2,pos2-pos1,t2,t3);
88     root=merge(t1, t2);
89 }
90 pnode findPos(pnode t, int pos) {
91     if(!t) return 0;
92     if(pos <= size(t->l)) return findPos(t->l, pos);
93     return findPos(t->r, pos - 1 - size(t->l));
94 }
95 Key &operator[](int pos){return findPos(root, pos)->key;}//ojito
96 };

```

2.11. Gain-Cost Set

```

1 //esta estructura mantiene pairs(beneficio, costo)
2 //de tal manera que en el set quedan ordenados
3 //por beneficio Y COSTO creciente. (va borrando los que no son optimos)
4 struct V{
5     int gain, cost;
6     bool operator<(const V &b)const{return gain<b.gain;}
7 };
8 set<V> s;
9 void add(V x){
10     set<V>::iterator p=s.lower_bound(x);//primer elemento mayor o igual
11     if(p!=s.end() && p->cost <= x.cost) return;//ya hay uno mejor
12     p=s.upper_bound(x);//primer elemento mayor
13     if(p!=s.begin()){//borro todos los peores (<=beneficio y >=costo)
14         --p;//ahora es ultimo elemento menor o igual
15         while(p->cost >= x.cost){
16             if(p==s.begin()){s.erase(p); break;}
17             s.erase(p--);
18         }
19     }
20     s.insert(x);
21 }
22 int get(int gain){//minimo costo de obtener tal ganancia
23     set<V>::iterator p=s.lower_bound((V){gain, 0});
24     return p==s.end()? INF : p->cost;}

```

3. Algos

3.1. Longest Increasing Subsequence

```

1 //Para non-increasing, cambiar comparaciones y revisar busq binaria
2 //Given an array, paint it in the least number of colors so that each
3 //color turns to a non-increasing subsequence.
4 //Solution:Min number of colors=Length of the longest increasing
5 //subsequence
6 int N, a[MAXN]; //secuencia y su longitud
7 int d[MAXN+1]; //d[i]=ultimo valor de la subsecuencia de tamaño i
8 int p[MAXN]; //padres
9 vector<int> R; //respuesta
10 void rec(int i){
11     if(i==0) return;
12     R.push_back(a[i]);
13     rec(p[i]);
14 }
15 int lis(){//O(nlogn)
16     d[0] = 0; //ii(-INF, -1); for(i, N) d[i+1]=ii(INF, -1);
17     for(i, N){
18         int j = upper_bound(d, d+N+1, ii(a[i], INF))-d;
19         if (d[j-1].first < a[i] && a[i] < d[j].first){
20             p[i]=d[j-1].second;
21             d[j] = ii(a[i], i);
22         }
23     }
24     R.clear();
25     dfor(i, N+1) if(d[i].first!=INF){
26         rec(d[i].second); //reconstruir
27         reverse(R.begin(), R.end());
28         return i; //longitud
29     }
30 }

```

3.2. Manacher

```

1 int d1[MAXN]; //d1[i]=long del maximo palindromo impar con centro en i
2 int d2[MAXN]; //d2[i]=analogo pero para longitud par
3 //0 1 2 3 4
4 //a a b c c <--d1[2]=3
5 //a a b b <--d2[2]=2 (están uno antes)

```



```

6 void manacher(){
7     int l=0, r=-1, n=sz(s);
8     forn(i, n){
9         int k=(i>r? 1 : min(d1[l+r-i], r-i));
10        while(i+k<n && i-k>=0 && s[i+k]==s[i-k]) ++k;
11        d1[i] = k--;
12        if(i+k > r) l=i-k, r=i+k;
13    }
14    l=0, r=-1;
15    forn(i, n){
16        int k=(i>r? 0 : min(d2[l+r-i+1], r-i+1))+1;
17        while(i+k-1<n && i-k>=0 && s[i+k-1]==s[i-k]) k++;
18        d2[i] = --k;
19        if(i+k-1 > r) l=i-k, r=i+k-1;
20    }

```

4. Strings

4.1. KMP

```

1 string T;//cadena donde buscar(what)
2 string P;//cadena a buscar(what)
3 int b[MAXLEN];//back table
4 void kmppre(){//by gabina with love
5     int i =0, j=-1; b[0]=-1;
6     while(i<sz(P)){
7         while(j>=0 && P[i] != P[j]) j=b[j];
8         i++, j++;
9         b[i] = j;
10    }
11 }
12
13 void kmp(){
14     int i=0, j=0;
15     while(i<sz(T)){
16         while(j>=0 && T[i]!=P[j]) j=b[j];
17         i++, j++;
18         if(j==sz(P)){
19             printf("P is found at index %d in T\n", i-j);
20             j=b[j];
21         }
22     }
23 }

```

4.2. Trie

```

1 struct trie{
2     map<char, trie> m;
3     void add(const string &s, int p=0){
4         if(s[p]) m[s[p]].add(s, p+1);
5     }
6     void dfs(){
7         //Do stuff
8         forall(it, m)
9             it->second.dfs();
10    }
11 };

```

4.3. Suffix Array (corto, nlog2n)

```

1 pair<int, int> sf[MAXN];
2 bool comp(int lhs, int rhs) {return sf[lhs] < sf[rhs];}
3 struct SuffixArray {
4     //sa guarda los indices de los sufijos ordenados
5     int sa[MAXN], r[MAXN];
6     void init(const char *a, int n) {
7         forn(i, n) r[i] = a[i];
8         for(int m = 1; m < n; m <= 1) {
9             forn(i, n) sa[i]=i, sf[i] = make_pair(r[i], i+m<n? r[i+m]:-1);
10            stable_sort(sa, sa+n, comp);
11            r[sa[0]] = 0;
12            forr(i, 1, n) r[sa[i]]= sf[sa[i]] != sf[sa[i] - 1] ? i : r[sa[i-1]];
13        }
14    }
15 } sa;

```

4.4. Suffix Array (largo, nlogn)

```

1 #define MAX_N 1000
2 #define rBOUND(x) (x<n? r[x] : 0)
3 //sa will hold the suffixes in order.
4 int sa[MAX_N], r[MAX_N], n;
5 string s; //input string, n=sz(s)
6
7 void countingSort(int k){
8     int f[MAX_N], tmpsa[MAX_N];
9     zero(f);

```

```

10  forn(i, n) f[rBOUND(i+k)]++;
11  int sum=0;
12  forn(i, max(255, n)){
13      int t=f[i]; f[i]=sum; sum+=t;}
14  forn(i, n)
15      tmpsa[f[rBOUND(sa[i]+k)]]+=sa[i];
16  memcpy(sa, tmpsa, sizeof(sa));
17  }
18  void constructsa(){//O(n log n)
19      n=sz(s);
20      forn(i, n) sa[i]=i, r[i]=s[i];
21      for(int k=1; k<n; k<=1){
22          countingSort(k), countingSort(0);
23          int rank, tmpr[MAX_N];
24          tmpr[sa[0]]=rank=0;
25          forr(i, 1, n)
26              tmpr[sa[i]]=(r[sa[i]]==r[sa[i-1]] && r[sa[i]+k]==r[sa[i-1]+k])?
                  rank : ++rank;
27          memcpy(r, tmpr, sizeof(r));
28          if(r[sa[n-1]]==n-1) break;
29      }
30  }
31  void print(){//for debug
32      forn(i, n)
33          cout << i << ' ' <<
34          s.substr(sa[i], s.find( '$', sa[i])-sa[i]) << endl;}

```

4.5. String Matching With Suffix Array

```

1  //returns (lowerbound, upperbound) of the search
2  ii stringMatching(string P){ //O(sz(P)lgn)
3      int lo=0, hi=n-1, mid=lo;
4      while(lo<hi){
5          mid=(lo+hi)/2;
6          int res=s.compare(sa[mid], sz(P), P);
7          if(res>=0) hi=mid;
8          else lo=mid+1;
9      }
10     if(s.compare(sa[lo], sz(P), P)!=0) return ii(-1, -1);
11     ii ans; ans.fst=lo;
12     lo=0, hi=n-1, mid;
13     while(lo<hi){
14         mid=(lo+hi)/2;

```

```

15     int res=s.compare(sa[mid], sz(P), P);
16     if(res>0) hi=mid;
17     else lo=mid+1;
18 }
19 if(s.compare(sa[hi], sz(P), P)!=0) hi--;
20 ans.snd=hi;
21 return ans;
22 }

```

4.6. LCP (Longest Common Prefix)

```

1  //Calculates the LCP between consecutives suffixes in the Suffix Array.
2  //LCP[i] is the length of the LCP between sa[i] and sa[i-1]
3  int LCP[MAX_N], phi[MAX_N], PLCP[MAX_N];
4  void computeLCP(){//O(n)
5      phi[sa[0]]=-1;
6      forr(i, 1, n) phi[sa[i]]=sa[i-1];
7      int L=0;
8      forn(i, n){
9          if(phi[i]==-1) {PLCP[i]=0; continue;}
10         while(s[i+L]==s[phi[i]+L]) L++;
11         PLCP[i]=L;
12         L=max(L-1, 0);
13     }
14     forn(i, n) LCP[i]=PLCP[sa[i]];
15 }

```

4.7. Corasick

```

1
2  struct trie{
3      map<char, trie> next;
4      trie* tran[256]; //transiciones del automata
5      int idhoja, szhoja; //id de la hoja o 0 si no lo es
6      //link lleva al sufijo mas largo, nxthoja lleva al mas largo pero que
        es hoja
7      trie *padre, *link, *nxthoja;
8      char pch; //caracter que conecta con padre
9      trie(): tran(), idhoja(), padre(), link() {}
10     void insert(const string &s, int id=1, int p=0){ //id>0!!!
11         if(p<sz(s)){
12             trie &ch=next[s[p]];
13             tran[(int)s[p]]=&ch;
14             ch.padre=this, ch.pch=s[p];

```

```

15     ch.insert(s, id, p+1);
16 }
17 else idhoja=id, szhoja=sz(s);
18 }
19 trie* get_link() {
20     if(!link){
21         if(!padre) link=this;//es la raiz
22         else if(!padre->padre) link=padre;//hijo de la raiz
23         else link=padre->get_link()->get_tran(pch);
24     }
25     return link;
26 }
27 trie* get_tran(int c) {
28     if(!tran[c])
29         tran[c] = !padre? this : this->get_link()->get_tran(c);
30     return tran[c];
31 }
32 trie *get_nxthoja(){
33     if(!nxthoja) nxthoja = get_link()->idhoja? link : link->nxthoja;
34     return nxthoja;
35 }
36 void print(int p){
37     if(idhoja)
38         cout << "found_" << idhoja << "_at_position_" << p-szhoja << endl
39         ;
40     if(get_nxthoja()) get_nxthoja()->print(p);
41 }
42 void matching(const string &s, int p=0){
43     print(p);
44     if(p<sz(s)) get_tran(s[p])->matching(s, p+1);

```

5. Geometría

#define EPS 1e-9

5.1. Punto

```

1 struct pto{
2     tipo x, y;
3     pto(tipo x=0, tipo y=0):x(x),y(y){}
4     pto operator+(pto a){return pto(x+a.x, y+a.y);}
5     pto operator-(pto a){return pto(x-a.x, y-a.y);}
6     pto operator+(tipo a){return pto(x+a, y+a);}

```

```

7     pto operator*(tipo a){return pto(x*a, y*a);}
8     pto operator/(tipo a){return pto(x/a, y/a);}
9     //dot product, producto interno:
10    tipo operator*(pto a){return x*a.x+y*a.y;}
11    //module of the cross product or vectorial product:
12    //if a is less than 180 clockwise from b, a^b>0
13    tipo operator^(pto a){return x*a.y-y*a.x;}
14    //returns true if this is at the left side of line qr
15    bool left(pto q, pto r){return ((q-*this)^(r-*this))>0;}
16    bool operator<(const pto &a) const{return x<a.x || (abs(x-a.x)<EPS &&
17        y<a.y);}
18    bool operator==(pto a){return abs(x-a.x)<EPS && abs(y-a.y)<EPS;}
19    double norm(){return sqrt(x*x+y*y);}
20    tipo norm_sq(){return x*x+y*y;}
21 };
22 double dist(pto a, pto b){return (b-a).norm();}
23 typedef pto vec;
24 double angle(pto a, pto o, pto b){
25     vec oa=a-o, ob=b-o;
26     return acos((oa*ob) / sqrt(oa.norm_sq()*ob.norm_sq()));}
27
28 //rotate p by theta rads CCW w.r.t. origin (0,0)
29 pto rotate(pto p, double theta){
30     return pto(p.x*cos(theta)-p.y*sin(theta),
31         p.x*sin(theta)+p.y*cos(theta));
32 }

```

5.2. Line

```

1 struct line{
2     line() {}
3     double a,b,c;//Ax+By=C
4     //pto MUST store float coordinates!
5     line(double a, double b, double c):a(a),b(b),c(c){}
6     line(pto p, pto q): a(q.y-p.y), b(p.x-q.x), c(a*p.x+b*p.y) {}
7 };
8 bool parallels(line l1, line l2){return abs(l1.a*l2.b-l2.a*l1.b)<EPS;}
9 pto inter(line l1, line l2){//intersection
10    double det=l1.a*l2.b-l2.a*l1.b;
11    if(abs(det)<EPS) return pto(INF, INF);//parallels
12    return pto(l2.b*l1.c-l1.b*l2.c, l1.a*l2.c-l2.a*l1.c)/det;
13 }

```

5.3. Segment

```

1 struct segm{
2   pto s,f;
3   segm(pto s, pto f):s(s), f(f) {}
4   pto closest(pto p) {//use for dist to point
5     double l2 = dist_sq(s, f);
6     if(l2==0.) return s;
7     double t = ((p-s)*(f-s))/l2;
8     if (t<0.) return s;//not write if is a line
9     else if(t>1.)return f;//not write if is a line
10    return s+((f-s)*t);
11  }
12  bool inside(pto p){
13  return ((s-p)^(f-p))==0 && min(s, f)<*this&&*this<max(s, f);}
14 };
15
16 bool insidebox(pto a, pto b, pto p) {
17   return (a.x-p.x)*(p.x-b.x)>-EPS && (a.y-p.y)*(p.y-b.y)>-EPS;
18 }
19 pto inter(segm s1, segm s2){
20   pto r=inter(line(s1.s, s1.f), line(s2.s, s2.f));
21   if(insidebox(s1.s,s1.f,p) && insidebox(s2.s,s2.f,p))
22     return r;
23   return pto(INF, INF);
24 }

```

5.4. Rectangle

```

1 struct rect{
2   //lower-left and upper-right corners
3   pto lw, up;
4 };
5 //returns if there's an intersection and stores it in r
6 bool inter(rect a, rect b, rect &r){
7   r.lw=pto(max(a.lw.x, b.lw.x), max(a.lw.y, b.lw.y));
8   r.up=pto(min(a.up.x, b.up.x), min(a.up.y, b.up.y));
9   //check case when only a edge is common
10  return r.lw.x<r.up.x && r.lw.y<r.up.y;
11 }

```

5.5. Polygon Area

```

1 double area(vector<tipo> &p){//0(sz(p))

```

```

2   double area=0;
3   forn(i, sz(p)) area+=p[i]^p[(i+1)%sz(p)];
4   //if points are in clockwise order then area is negative
5   return abs(area)/2;
6 }
7 //Area ellipse = M_PI*a*b where a and b are the semi axis lengths
8 //Area triangle = sqrt(s*(s-a)(s-b)(s-c)) where s=(a+b+c)/2

```

5.6. Circle

```

1 vec perp(vec v){return vec(-v.y, v.x);}
2 line bisector(pto x, pto y){
3   line l=line(x, y); pto m=(x+y)/2;
4   return line(-l.b, l.a, -l.b*m.x+l.a*m.y);
5 }
6 struct Circle{
7   pto o;
8   double r;
9   //circle determined by three points, uses line
10  Circle(pto x, pto y, pto z){
11    o=inter(bisector(x, y), bisector(y, z));
12    r=dist(o, x);
13  }
14  pair<pto, pto> ptosTang(pto p){
15    pto m=(p+o)/2;
16    tipo d=dist(o, m);
17    tipo a=r*r/(2*d);
18    tipo h=sqrt(r*r-a*a);
19    pto m2=o+(m-o)*a/d;
20    vec per=perp(m-o)/d;
21    return mkp(m2-per*h, m2+per*h);
22  }
23 };
24 //finds the center of the circle containing p1 and p2 with radius r
25 //as there may be two solutions swap p1, p2 to get the other
26 bool circle2PtsRad(pto p1, pto p2, double r, pto &c){
27   double d2=(p1-p2).norm_sq(), det=r*r/d2-0.25;
28   if(det<0) return false;
29   c=(p1+p2)/2+perp(p2-p1)*sqrt(det);
30   return true;
31 }

```

5.7. Point in Poly

```

1 //checks if v is inside of P, using ray casting
2 //works with convex and concave.
3 //excludes boundaries, handle it separately using segment.inside()
4 bool inPolygon(pto v, vector<pto>& P) {
5     bool c = false;
6     forn(i, sz(P)){
7         int j=(i+1)%sz(P);
8         if((P[j].y>v.y) != (P[i].y > v.y) &&
9         (v.x < (P[i].x - P[j].x) * (v.y-P[j].y) / (P[i].y - P[j].y) + P[j].x))
10             c = !c;
11     }
12     return c;
13 }

```

5.8. Convex Check CHECK

```

1 bool isConvex(vector<int> &p){//O(N)
2     int N=sz(p);
3     if(N<3) return false;
4     bool isLeft=p[0].left(p[1], p[2]);
5     forr(i, 1, N)
6         if(p[i].left(p[(i+1)%N], p[(i+2)%N])!=isLeft)
7             return false;
8     return true; }

```

5.9. Convex Hull

```

1 //stores convex hull of P in S, CCW order
2 void CH(vector<pto>& P, vector<pto> &S){
3     S.clear();
4     sort(P.begin(), P.end());
5     forn(i, sz(P)){
6         while(sz(S)>= 2 && S[sz(S)-1].left(S[sz(S)-2], P[i])) S.pop_back();
7         S.pb(P[i]);
8     }
9     S.pop_back();
10    int k=sz(S);
11    dforn(i, sz(P)){
12        while(sz(S) >= k+2 && S[sz(S)-1].left(S[sz(S)-2], P[i])) S.pop_back
13            ();
14        S.pb(P[i]);
15    }
16    S.pop_back();
17 }

```

5.10. Cut Polygon

```

1 //cuts polygon Q along the line ab
2 //stores the left side (swap a, b for the right one) in P
3 void cutPolygon(pto a, pto b, vector<pto> Q, vector<pto> &P){
4     P.clear();
5     forn(i, sz(Q)){
6         double left1=(b-a)^(Q[i]-a), left2=(b-a)^(Q[(i+1)%sz(Q)]-a);
7         if(left1>=0) P.pb(Q[i]);
8         if(left1*left2<0)
9             P.pb(inter(line(Q[i], Q[(i+1)%sz(Q)]), line(a, b)));
10    }
11 }

```

5.11. Bresenham

```

1 //plot a line approximation in a 2d map
2 void bresenham(pto a, pto b){
3     pto d=b-a; d.x=abs(d.x), d.y=abs(d.y);
4     pto s(a.x<b.x? 1: -1, a.y<b.y? 1: -1);
5     int err=d.x-d.y;
6     while(1){
7         m[a.x][a.y]=1;//plot
8         if(a==b) break;
9         int e2=2*err;
10        if(e2 > -d.y){
11            err-=d.y, a.x+=s.x;
12        }
13        if(e2 < d.x){
14            err+= d.x, a.y+= s.y;
15        }
16    }
17 }

```

5.12. Rotate Matrix

```

1 //rotates matrix t 90 degrees clockwise
2 //using auxiliary matrix t2(faster)
3 void rotate(){
4     forn(x, n) forn(y, n)
5         t2[n-y-1][x]=t[x][y];
6     memcpy(t, t2, sizeof(t));
7 }

```

6. Math

6.1. Identidades

$$\sum_{i=0}^n \binom{n}{i} = 2^n$$

$$\sum_{i=0}^n i \binom{n}{i} = n * 2^{n-1}$$

$$\sum_{i=m}^n i = \frac{n(n+1)}{2} - \frac{m(m-1)}{2} = \frac{(n+1-m)(n+m)}{2}$$

$$\sum_{i=0}^n i = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

$$\sum_{i=0}^n i(i-1) = \frac{8}{6} \left(\frac{n}{2}\right) \left(\frac{n}{2} + 1\right) (n+1)$$

(doubles) → Sino ver caso impar y par

$$\sum_{i=0}^n i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} = \left[\sum_{i=1}^n i\right]^2$$

$$\sum_{i=0}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30}$$

$$\sum_{i=0}^n i^p = \frac{(n+1)^{p+1}}{p+1} + \sum_{k=1}^p \frac{B_k}{p-k+1} \binom{p}{k} (n+1)^{p-k+1}$$

6.2. Combinatorio

```

1 | forn(i, MAXN+1){ //comb[i][k]=i tomados de a k
2 |   comb[i][0]=comb[i][i]=1;
3 |   forr(k, 1, i) comb[i][k]=(comb[i-1][k]+comb[i-1][k-1])%MOD;
4 | }
5 |
6 | ll lucas (ll n, ll k, int p){ //Calcula (n,k)%p teniendo comb[p][p]
   |   precalculado.
7 |   ll aux = 1;
8 |   while (n + k){

```

```

9 |   aux = (aux * comb[n%p][k%p]) %p;
10 |   n/=p, k/=p;
11 | }
12 | return aux;
13 | }

```

6.3. Exp. de Numeros Mod.

```

1 | ll expmod (ll b, ll e, ll m){ //O(log b)
2 |   if(!e) return 1;
3 |   ll q= expmod(b,e/2,m); q=(q*q)%m;
4 |   return e%2? (b * q)%m : q;
5 | }

```

6.4. Exp. de Matrices y Fibonacci en log(n)

```

1 | struct M22{ // la b|
2 |   tipo a,b,c,d; // lc dl
3 |   M22 operator*(const M22 &p) const {
4 |     return (M22){a*p.a+b*p.c, a*p.b+b*p.d, c*p.a+d*p.c,c*p.b+d*p.d};}
5 | };
6 | M22 operator^(const M22 &p, int n){
7 |   if(!n) return (M22){1, 0, 0, 1}; //identidad
8 |   M22 q=p^(n/2); q=q*q;
9 |   return n%2? p * q : q;}
10 |
11 | ll fibo(ll n){ //calcula el fibonacci enesimo
12 |   M22 mat=(M22){0, 1, 1, 1}^n;
13 |   return mat.a*f0+mat.b*f1; //f0 y f1 son los valores iniciales
14 | }

```

6.5. Teorema Chino del Resto

$$y = \sum_{j=1}^n (x_j * (\prod_{i=1, i \neq j}^n m_i)_{m_j}^{-1} * \prod_{i=1, i \neq j}^n m_i)$$

6.6. Funciones de primos

```

1 | ll numPrimeFactors (ll n){
2 |   ll rta = 0;
3 |   map<ll,ll> f=fact(n);
4 |   forall(it, f) rta += it->second;

```

```

5   return rta;
6 }
7
8 ll numDiffPrimeFactors (ll n){
9     ll rta = 0;
10    map<ll,ll> f=fact(n);
11    forall(it, f) rta += 1;
12    return rta;
13 }
14
15 ll sumPrimeFactors (ll n){
16     ll rta = 0;
17     map<ll,ll> f=fact(n);
18     forall(it, f) rta += it->first;
19     return rta;
20 }
21
22 ll numDiv (ll n){
23     ll rta = 1;
24     map<ll,ll> f=fact(n);
25     forall(it, f) rta *= (it->second + 1);
26     return rta;
27 }
28
29 ll sumDiv (ll n){
30     ll rta = 1;
31     map<ll,ll> f=fact(n);
32     forall(it, f) rta *= ((ll)pow((double)it->first, it->second + 1.0)-1)
33         / (it->first-1);
34     return rta;
35 }
36
37 ll eulerPhi (ll n){ // con criba: O(lg n)
38     ll rta = n;
39     map<ll,ll> f=fact(n);
40     forall(it, f) rta -= rta / it->first;
41     return rta;
42 }
43
44 ll eulerPhi2 (ll n){ // O (sqrt n)
45     ll r = n;
46     forr (i,2,n+1){
47         if ((ll)i*i > n)

```

```

47         break;
48         if (n % i == 0){
49             while (n%i == 0) n/=i;
50             r -= r/i;
51         }}
52     if (n != 1)
53         r-= r/n;
54     return r;
55 }

```

6.7. Phollard's Rho (rolando)

```

1 ll mulmod (ll a, ll b, ll c) { //returns (a*b)%c, and minimize overflow
2     ll x = 0, y = a%c;
3     while (b > 0){
4         if (b % 2 == 1) x = (x+y) % c;
5         y = (y*2) % c;
6         b /= 2;
7     }
8     return x % c;
9 }
10
11 bool es_primo_prob (ll n, int a)
12 {
13     ll s = 0, d = n-1;
14     while (d % 2 == 0) s++, d/=2;
15
16     ll x = expmod(a,d,n);
17     if ((x == 1) || (x+1 == n)) return true;
18
19     forn (i, s-1){
20         x = (x*x)%n;
21         if (x == 1) return false;
22         if (x+1 == n) return true;
23     }
24     return false;
25 }
26
27 bool miller_rabin (ll n){ //devuelve true si n es primo
28     const int ar[] = {2,3,5,7,11,13,17,19,23};
29     forn (j,9)
30         if (!es_primo_prob(n,ar[j]))
31             return false;

```

```

32 return true;
33 }
34
35 ll pollard_rho (ll n){
36     int i = 0, k = 2;
37     ll x = 3, y = 3;
38     while (1){
39         i++;
40         x = (mulmod (x,x,n) + n - 1) % n;
41         ll d = gcd (abs(y-x), n);
42         if (d != 1 && d != n) return d;
43         if (i == k) y = x, k*=2;
44     }
45 }

```

6.8. Criba

```

1 #define MAXP 80000 //no necesariamente primo
2 int criba[MAXP+1];
3 vector<int> primos;
4 void buscarprimos(){
5     int sq=sqrt(MAXP)+1;
6     forr(p, 2, MAXP+1) if(!criba[p]){
7         primos.push_back(p);
8         if(p<=sq)
9             for(int m=p*p; m<=MAXP; m+=p)//borro los multiplos de p
10                 if(!criba[m])criba[m]=p;
11     }
12 }

```

6.9. Factorizacion

Sea $n = \prod p_i^{k_i}$, fact(n) genera un map donde a cada p_i le asocia su k_i

```

1 //factoriza bien numeros hasta MAXP^2
2 map<ll,ll> fact(ll n){ //0 (cant primos)
3     map<ll,ll> ret;
4     forall(p, primos){
5         while(!(n%p)){
6             ret[*p]++; //divisor found
7             n/=p;
8         }
9     }
10     if(n>1) ret[n]++;
11     return ret;

```

```

12 }
13
14 //factoriza bien numeros hasta MAXP
15 map<ll,ll> fact2(ll n){ //0 (lg n)
16     map<ll,ll> ret;
17     while (criba[n]){
18         ret[criba[n]]++;
19         n/=criba[n];
20     }
21     if(n>1) ret[n]++;
22     return ret;
23 }

```

6.10. GCD

```

1 tipo gcd(tipo a, tipo b){return a?gcd(b %a, a):b;}

```

6.11. LCM

```

1 tipo lcm(tipo a, tipo b){return a / gcd(a,b) * b;}

```

6.12. Inversos

```

1 #define MAXMOD 15485867
2 ll inv[MAXMOD]; //inv[i]*i=1 mod MOD
3 void calc(int p){//0(p)
4     inv[1]=1;
5     forr(i, 2, p) inv[i]= p-((p/i)*inv[p%i])%p;
6 }
7 int inverso(int x){//0(log x)
8     return expmod(x, eulerphi(MOD)-2); //si mod no es primo(sacar a mano)
9     return expmod(x, MOD-2); //si mod es primo
10 }

```

6.13. Simpson

```

1 double integral(double a, double b, int n=10000) { //0(n), n=cantdiv
2     double area=0, h=(b-a)/n, fa=f(a), fb;
3     forn(i, n){
4         fb=f(a+h*(i+1));
5         area+=fa+ 4*f(a+h*(i+0.5)) +fb, fa=fb;
6     }
7     return area*h/6.;}

```


6.14. Fraction

```

1 struct frac{
2     tipo p,q;
3     frac(tipo p=0, tipo q=1):p(p),q(q) {norm();}
4     tipo mcd(tipo a, tipo b){return a?mcd(b %a, a):b;}
5     void norm(){
6         tipo a = mcd(p,q);
7         if(a) p/=a, q/=a;
8         else q=1;
9         if (q<0) q=-q, p=-p;}
10    frac operator+(const frac& o){
11        tipo a = mcd(q,o.q);
12        return frac(p*(o.q/a)+o.p*(q/a), q*(o.q/a));}
13    frac operator-(const frac& o){
14        tipo a = mcd(q,o.q);
15        return frac(p*(o.q/a)-o.p*(q/a), q*(o.q/a));}
16    frac operator*(frac o){
17        tipo a = mcd(q,o.p), b = mcd(o.q,p);
18        return frac((p/b)*(o.p/a), (q/a)*(o.q/b));}
19    frac operator/(frac o){
20        tipo a = mcd(q,o.q), b = mcd(o.p,p);
21        return frac((p/b)*(o.q/a),(q/a)*(o.p/b));}
22    bool operator<(const frac &o) const{return p*o.q < o.p*q;}
23    bool operator==(frac o){return p==o.p&&q==o.q;}
24 };

```

6.15. Polinomio

```

1 #define MAX_GR 20
2 struct poly {
3     tipo p[MAX_GR]; //guarda los coeficientes del polinomio
4     poly(){zero(p);}
5     int gr(){//calculates grade of the polynomial
6         dforn(i,MAX_GR) if(p[i]) return i;
7         return 0; }
8     bool isnull() {return gr()==0 && !p[0];}
9     poly operator+(poly b) {// - is analogous
10        poly c=THIS;
11        forn(i,MAX_GR) c.p[i]+=b.p[i];
12        return c;
13    }
14    poly operator*(poly b) {

```

```

15    poly c;
16    forn(i,MAX_GR) forn(k,i+1) c.p[i]+=p[k]*b.p[i-k];
17    return c;
18 }
19 tipo eval(tipo v) {
20     tipo sum = 0;
21     dforn(i, MAX_GR) sum=sum*v + p[i];
22     return sum;
23 }
24 //the following function generates the roots of the polynomial
25 //it can be easily modified to return float roots
26 set<tipo> roots(){
27     set<tipo> roots;
28     tipo a0 = abs(p[0]), an = abs(p[gr()]);
29     vector<tipo> ps,qs;
30     forr(p,1,sqrt(a0)+1) if (a0%p==0) ps.pb(p),ps.pb(a0/p);
31     forr(q,1,sqrt(an)+1) if (an%q==0) qs.pb(q),qs.pb(an/q);
32     forall(pt,ps)
33         forall(qt,qs) if ( (*pt) % (*qt)==0 ) {
34             tipo root = abs((*pt) / (*qt));
35             if (eval(root)==0) roots.insert(root);
36         }
37     return roots;
38 }
39 };
40 //the following functions allows parsing an expression like
41 //34+150+4*45
42 //into a polynomial(el numero en funcion de la base)
43 #define LAST(s) (sz(s)? s[sz(s)-1] : 0)
44 #define POP(s) s.erase(--s.end());
45 poly D(string &s) {
46     poly d;
47     for(int i=0; isdigit(LAST(s)); i++) d.p[i]=LAST(s)-'0', POP(s);
48     return d;}
49
50 poly T(string &s) {
51     poly t=D(s);
52     if (LAST(s)=='*'){POP(s); return T(s)*t;}
53     return t;
54 }
55 //main function, call this to parse
56 poly E(string &s) {
57     poly e=T(s);

```

```

58 if (LAST(s)=='+'){POP(s); return E(s)+e;}
59 return e;
60 }

```

7. Grafos

7.1. Dijkstra

```

1 #define INF 1e9
2 int N;
3 #define MAX_V 250001
4 vector<ii> G[MAX_V];
5 //To add an edge use
6 #define add(a, b, w) G[a].pb(mkp(w, b))
7
8 ll dijkstra(int s, int t){//O(|E| log |V|)
9     priority_queue<ii, vector<ii>, greater<ii> > Q;
10     vector<ll> dist(N, INF); vector<int> dad(N, -1);
11     Q.push(mkp(0, s)); dist[s] = 0;
12     while(sz(Q)){
13         ii p = Q.top(); Q.pop();
14         if(p.snd == t) break;
15         forall(it, G[p.snd])
16             if(dist[p.snd]+it->fst < dist[it->snd]){
17                 dist[it->snd] = dist[p.snd] + it->fst;
18                 dad[it->snd] = p.snd;
19                 Q.push(mkp(dist[it->snd], it->snd));
20             }
21     }
22     return dist[t];
23     if(dist[t]<INF)//path generator
24     for(int i=t; i!=-1; i=dad[i])
25         printf("%d%c", i, (i==s?'\\n':' '));
26 }

```

7.2. Bellman-Ford

```

1 vector<ii> G[MAX_N]; //ady. list with pairs (weight, dst)
2 int dist[MAX_N];
3 void bford(int src){//O(VE)
4     dist[src]=0;
5     forn(i, N-1) forn(j, N) if(dist[j]!=INF) forall(it, G[j])
6         dist[it->snd]=min(dist[it->snd], dist[j]+it->fst);

```

```

7 }
8
9 bool hasNegCycle(){
10     forn(j, N) if(dist[j]!=INF) forall(it, G[j])
11         if(dist[it->snd]>dist[j]+it->fst) return true;
12     //inside if: all points reachable from it->snd will have -INF distance
13     (do bfs)
14     return false;
15 }

```

7.3. Floyd-Warshall

```

1 //G[i][j] contains weight of edge (i, j) or INF
2 //G[i][i]=0
3 int G[MAX_N][MAX_N];
4 void floyd(){//O(N^3)
5     forn(k, N) forn(i, N) if(G[i][k]!=INF) forn(j, N) if(G[k][j]!=INF)
6         G[i][j]=min(G[i][j], G[i][k]+G[k][j]);
7 }
8 bool inNegCycle(int v){
9     return G[v][v]<0;}
10 //checks if there's a neg. cycle in path from a to b
11 bool hasNegCycle(int a, int b){
12     forn(i, N) if(G[a][i]!=INF && G[i][i]<0 && G[i][b]!=INF)
13         return true;
14     return false;
15 }

```

7.4. Kruskal

```

1 struct Ar{int a,b,w;};
2 bool operator<(const Ar& a, const Ar &b){return a.w<b.w;}
3 vector<Ar> E;
4 ll kruskal(){
5     ll cost=0;
6     sort(E.begin(), E.end()); //ordenar aristas de menor a mayor
7     uf.init(n);
8     forall(it, E){
9         if(uf.comp(it->a)!=uf.comp(it->b)){//si no estan conectados
10             uf.unir(it->a, it->b); //conectar
11             cost+=it->w;
12         }
13     }
14     return cost;

```

7.5. Prim

```

15 }

1 bool taken[MAXN];
2 priority_queue<ii, vector<ii>, greater<ii> > pq; //min heap
3 void process(int v){
4     taken[v]=true;
5     forall(e, G[v])
6         if(!taken[e->second]) pq.push(*e);
7 }
8
9 ll prim(){
10     zero(taken);
11     process(0);
12     ll cost=0;
13     while(sz(pq)){
14         ii e=pq.top(); pq.pop();
15         if(!taken[e.second]) cost+=e.first, process(e.second);
16     }
17     return cost;
18 }

```

7.6. 2-SAT + Tarjan SCC

```

1 //We have a vertex representing a var and other for his negation.
2 //Every edge stored in G represents an implication. To add an equation
3 //of the form a||b, use addor(a, b)
4 //MAX=max cant var, n=cant var
5 #define addor(a, b) (G[neg(a)].pb(b), G[neg(b)].pb(a))
6 vector<int> G[MAX*2];
7 //idx[i]=index assigned in the dfs
8 //lw[i]=lowest index(closer from the root) reachable from i
9 int lw[MAX*2], idx[MAX*2], qidx;
10 stack<int> q;
11 int qcmp, cmp[MAX*2];
12 //verdad[cmp[i]]=valor de la variable i
13 bool verdad[MAX*2+1];
14
15 int neg(int x) { return x>=n? x-n : x+n;}
16 void tjn(int v){
17     lw[v]=idx[v]=++qidx;
18     q.push(v), cmp[v]=-2;
19     forall(it, G[v]){

```

```

19         if(!idx[*it] || cmp[*it]==-2){
20             if(!idx[*it]) tjn(*it);
21             lw[v]=min(lw[v], lw[*it]);
22         }
23     }
24     if(lw[v]==idx[v]){
25         qcmp++;
26         int x;
27         do{x=q.top(); q.pop(); cmp[x]=qcmp;}while(x!=v);
28         verdad[qcmp]=(cmp[neg(v)]<0);
29     }
30 }
31 //remember to CLEAR G!!!
32 bool satisf(){//O(n)
33     memset(idx, 0, sizeof(idx)), qidx=0;
34     memset(cmp, -1, sizeof(cmp)), qcmp=0;
35     forn(i, n){
36         if(!idx[i]) tjn(i);
37         if(!idx[neg(i)]) tjn(neg(i));
38     }
39     forn(i, n) if(cmp[i]==cmp[neg(i)]) return false;
40     return true;
41 }

```

7.7. Articulation Points

```

1 int N;
2 vector<int> G[1000000];
3 //V[i]=node number(if visited), L[i]= lowest V[i] reachable from i
4 int qV, V[1000000], L[1000000], P[1000000];
5 void dfs(int v, int f){
6     L[v]=V[v]=++qV;
7     forall(it, G[v])
8         if(!V[*it]){
9             dfs(*it, v);
10            L[v] = min(L[v], L[*it]);
11            P[v] += L[*it]>=V[v];
12        }
13        else if(*it!=f)
14            L[v]=min(L[v], V[*it]);
15    }
16    int cantart(){ //O(n)
17        qV=0;

```

```

18 zero(V), zero(P);
19 dfs(1, 0); P[1]--;
20 int q=0;
21 forn(i, N) if(P[i]) q++;
22 return q;
23 }

```

7.8. Comp. Biconexas y Puentes

```

1 struct edge {
2     int u,v, comp;
3     bool bridge;
4 };
5 vector<edge> e;
6 void addEdge(int u, int v) {
7     G[u].pb(sz(e)), G[v].pb(sz(e));
8     e.pb((edge){u,v,-1,false});
9 }
10 //d[i]=id de la dfs
11 //b[i]=lowest id reachable from i
12 int d[MAXN], b[MAXN], t;
13 int nbc;//cant componentes
14 int comp[MAXN]; //comp[i]=cant comp biconexas a la cual pertenece i
15 void initDfs(int n) {
16     zero(G), zero(comp);
17     e.clear();
18     forn(i,n) d[i]=-1;
19     nbc = t = 0;
20 }
21 stack<int> st;
22 void dfs(int u, int pe) { //O(n + m)
23     b[u] = d[u] = t++;
24     comp[u] = (pe != -1);
25     forall(ne, G[u]) if (*ne != pe){
26         int v = e[*ne].u ^ e[*ne].v ^ u;
27         if (d[v] == -1) {
28             st.push(*ne);
29             dfs(v,*ne);
30             if (b[v] > d[u]){
31                 e[*ne].bridge = true; // bridge
32             }
33             if (b[v] >= d[u]){ // art
34                 int last;

```

```

35         do {
36             last = st.top(); st.pop();
37             e[last].comp = nbc;
38         } while (last != *ne);
39         nbc++;
40         comp[u]++;
41     }
42     b[u] = min(b[u], b[v]);
43 }
44 else if (d[v] < d[u]) { // back edge
45     st.push(*ne);
46     b[u] = min(b[u], d[v]);
47 }
48 }
49 }

```

7.9. LCA + Clim

```

1 //f[v][k] holds the 2^k father of v
2 //L[v] holds the level of v
3 int N, f[100001][20], L[100001];
4 void build(){ //f[i][0] must be filled previously, O(nlgn)
5     forn(k, 20-1) forn(i, N) f[i][k+1]=f[f[i][k]][k];}
6
7 #define lg(x) (31-__builtin_clz(x))//=floor(log2(x))
8
9 int climb(int a, int d){ //O(lgn)
10     if(!d) return a;
11     dforn(i, lg(L[a])+1)
12         if(1<<i<=d)
13             a=f[a][i], d-=1<<i;
14     return a;
15 }
16 int lca(int a, int b){ //O(lgn)
17     if(L[a]<L[b]) swap(a, b);
18     a=climb(a, L[a]-L[b]);
19     if(a==b) return a;
20     dforn(i, lg(L[a])+1)
21         if(f[a][i]!=f[b][i])
22             a=f[a][i], b=f[b][i];
23     return f[a][0];
24 }

```

7.10. Heavy Light Decomposition

```

1 int treesz[MAXN]; //cantidad de nodos en el subarbol del nodo v
2 int dad[MAXN]; //dad[v]=padre del nodo v
3 void dfs1(int v, int p=-1) { //pre-dfs
4     dad[v]=p;
5     treesz[v]=1;
6     forall(it, G[v]) if(*it!=p){
7         dfs1(*it, v);
8         treesz[v]+=treesz[*it];
9     }
10 }
11 int pos[MAXN], q; //pos[v]=posicion del nodo v en el recorrido de la dfs
12 //Las cadenas aparecen continuas en el recorrido!
13 int cantcad;
14 int homecad[MAXN]; //dada una cadena devuelve su nodo inicial
15 int cad[MAXN]; //cad[v]=cadena a la que pertenece el nodo
16 void heavylight(int v, int cur=-1){
17     if(cur==--1) homecad[cur=cantcad++]=v;
18     pos[v]=q++;
19     cad[v]=cur;
20     int mx=-1;
21     forn(i, sz(G[v])) if(G[v][i]!=dad[v]){
22         if(mx==-1 || treesz[G[v][mx]]<treesz[G[v][i]]) mx=i;
23         if(mx!=-1) heavylight(G[v][mx], cur);
24         forn(i, sz(G[v])) if(i!=mx && G[v][i]!=dad[v]){
25             heavylight(G[v][i], -1);
26         }
27     }
28     //ejemplo de obtener el maximo numero en el camino entre dos nodos
29     //RTA: max(query(low, u), query(low, v)), con low=lca(u, v)
30     //esta funcion va trepando por las cadenas
31     int query(int an, int v) { //O(logn)
32         //si estan en la misma cadena:
33         if(cad[an]==cad[v]) return rmq.get(pos[an], pos[v]+1);
34         return max(query(an, dad[homecad[cad[v]]]),
35             rmq.get(pos[homecad[cad[v]]], pos[v]+1));
36     }
37 }

```

8. Network Flow

8.1. Dinic

```

1 int nodes, src, dest;

```

```

2 int dist[MAX], q[MAX], work[MAX];
3
4 struct Edge {
5     int to, rev;
6     ll f, cap;
7     Edge(int to, int rev, ll f, ll cap) : to(to), rev(rev), f(f), cap(cap)
8     {}
9 };
10 vector<Edge> G[MAX];
11
12 // Adds bidirectional edge
13 void addEdge(int s, int t, ll cap){
14     G[s].push_back(Edge(t, G[t].size(), 0, cap));
15     G[t].push_back(Edge(s, G[s].size()-1, 0, 0));
16 }
17
18 bool dinic_bfs() {
19     fill(dist, dist + nodes, -1);
20     dist[src] = 0;
21     int qt = 0;
22     q[qt++] = src;
23     for (int qh = 0; qh < qt; qh++) {
24         int u = q[qh];
25         forall(e, G[u]){
26             int v = e->to;
27             if(dist[v]<0 && e->f < e->cap){
28                 dist[v]=dist[u]+1;
29                 q[qt++]=v;
30             }
31         }
32     }
33     return dist[dest] >= 0;
34 }
35
36 ll dinic_dfs(int u, ll f) {
37     if (u == dest) return f;
38     for (int &i = work[u]; i < (int) G[u].size(); i++) {
39         Edge &e = G[u][i];
40         if (e.cap <= e.f) continue;
41         int v = e.to;
42         if (dist[v] == dist[u] + 1) {
43             ll df = dinic_dfs(v, min(f, e.cap - e.f));

```

```

44     if (df > 0) {
45         e.f += df;
46         G[v][e.rev].f -= df;
47         return df;
48     }
49 }
50 }
51 return 0;
52 }
53
54 ll maxFlow(int _src, int _dest) { //O(V^2 E)
55     src = _src;
56     dest = _dest;
57     ll result = 0;
58     while (dinic_bfs()) {
59         fill(work, work + nodes, 0);
60         while (ll delta = dinic_dfs(src, INF))
61             result += delta;
62     }
63
64     // todos los nodos con dist[v] != -1 vs los que tienen dist[v] == -1
        forman el min cut

```

8.2. Konig

```

1 // asume que el dinic YA ESTA tirado
2 // asume que nodes-1 y nodes-2 son la fuente y destino
3 int match[maxnodes]; // match[v]=u si u-v esta en el matching, -1 si v
    no esta matcheado
4 int s[maxnodes]; // numero de la bfs del koning
5 queue<int> kq;
6 // s[e] %2==1 o si e esta en V1 y s[e]==-1-> lo agarras
7 void koning() { //O(n)
8     forn(v,nodes-2) s[v] = match[v] = -1;
9     forn(v,nodes-2) forall(it,g[v]) if (it->to < nodes-2 && it->f>0)
10         { match[v]=it->to; match[it->to]=v; }
11     forn(v,nodes-2) if (match[v]==-1) { s[v]=0; kq.push(v); }
12     while(!kq.empty()) {
13         int e = kq.front(); kq.pop();
14         if (s[e] %2==1) {
15             s[match[e]] = s[e]+1;
16             kq.push(match[e]);
17         } else {

```

```

18
19         forall(it,g[e]) if (it->to < nodes-2 && s[it->to]==-1) {
20             s[it->to] = s[e]+1;
21             kq.push(it->to);
22         }
23     }
24 }
25 }

```

8.3. Edmonds Karp's

```

1 #define MAX_V 1000
2 #define INF 1e9
3 //special nodes
4 #define SRC 0
5 #define SNK 1
6 map<int, int> G[MAX_V]; //limpiar esto
7 //To add an edge use
8 #define add(a, b, w) G[a][b]=w
9 int f, p[MAX_V];
10 void augment(int v, int minE){
11     if(v==SRC) f=minE;
12     else if(p[v] != -1){
13         augment(p[v], min(minE, G[p[v]][v]));
14         G[p[v]][v] -= f, G[v][p[v]] += f;
15     }
16 }
17 ll maxflow(){ //O(VE^2)
18     ll Mf=0;
19     do{
20         f=0;
21         char used[MAX_V]; queue<int> q; q.push(SRC);
22         zero(used), memset(p, -1, sizeof(p));
23         while(sz(q)){
24             int u=q.front(); q.pop();
25             if(u==SNK) break;
26             forall(it, G[u])
27                 if(it->snd>0 && !used[it->fst])
28                     used[it->fst]=true, q.push(it->fst), p[it->fst]=u;
29         }
30         augment(SNK, INF);
31         Mf+=f;
32     }while(f);

```

```

33 | return Mf;
34 | }

```

8.4. Push-Relabel

```

1 | #define MAX_V 1000
2 | int N; //valid nodes are [0...N-1]
3 | #define INF 1e9
4 | //special nodes
5 | #define SRC 0
6 | #define SNK 1
7 | map<int, int> G[MAX_V];
8 | //To add an edge use
9 | #define add(a, b, w) G[a][b]=w
10 | ll excess[MAX_V];
11 | int height[MAX_V], active[MAX_V], count[2*MAX_V+1];
12 | queue<int> Q;
13 | void enqueue(int v) {
14 |     if (!active[v] && excess[v] > 0) active[v]=true, Q.push(v); }
15 | void push(int a, int b) {
16 |     int amt = min(excess[a], ll(G[a][b]));
17 |     if(height[a] <= height[b] || amt == 0) return;
18 |     G[a][b]-=amt, G[b][a]+=amt;
19 |     excess[b] += amt, excess[a] -= amt;
20 |     enqueue(b);
21 | }
22 | void gap(int k) {
23 |     forn(v, N){
24 |         if (height[v] < k) continue;
25 |         count[height[v]]--;
26 |         height[v] = max(height[v], N+1);
27 |         count[height[v]]++;
28 |         enqueue(v);
29 |     }
30 | }
31 | void relabel(int v) {
32 |     count[height[v]]--;
33 |     height[v] = 2*N;
34 |     forall(it, G[v])
35 |         if(it->snd)
36 |             height[v] = min(height[v], height[it->fst] + 1);
37 |     count[height[v]]++;
38 |     enqueue(v);

```

```

39 | }
40 | ll maxflow() { //O(V^3)
41 |     zero(height), zero(active), zero(count), zero(excess);
42 |     count[0] = N-1;
43 |     count[N] = 1;
44 |     height[SRC] = N;
45 |     active[SRC] = active[SNK] = true;
46 |     forall(it, G[SRC]){
47 |         excess[SRC] += it->snd;
48 |         push(SRC, it->fst);
49 |     }
50 |     while(sz(Q)) {
51 |         int v = Q.front(); Q.pop();
52 |         active[v]=false;
53 |         forall(it, G[v]) push(v, it->fst);
54 |         if(excess[v] > 0)
55 |             count[height[v]] == 1? gap(height[v]):relabel(v);
56 |     }
57 |     ll mf=0;
58 |     forall(it, G[SRC]) mf+=G[it->fst][SNK];
59 |     return mf;
60 | }

```

9. Ayudamemoria

Límites

```

1 | #include <climits> //INT_MIN, LONG_MAX, ULLONG_MAX, etc.

```

Cant. decimales

```

1 | #include <iomanip>
2 | cout << setprecision(2) << fixed;

```

Rellenar con espacios(para justificar)

```

1 | #include <iomanip>
2 | cout << setfill('␣') << setw(3) << 2 << endl;

```

Leer hasta fin de línea

```

1 | #include <sstream>
2 | //hacer cin.ignore() antes de getline()
3 | while(getline(cin, line)){

```

```
4 |     istream is(line);
5 |     while(is >> X)
6 |         cout << X << "□";
7 |     cout << endl;
8 | }
```

Aleatorios

```
1 | #define RAND(a, b) (rand()%(b-a+1)+a)
2 | srand(time(NULL));
```

Doubles Comp.

```
1 | const double EPS = 1e-9;
2 | x == y <=> fabs(x-y) < EPS
3 | x > y <=> x > y + EPS
4 | x >= y <=> x > y - EPS
```

Límites

```
1 | #include <limits>
2 | numeric_limits<T>
3 |     ::max()
4 |     ::min()
5 |     ::epsilon()
```

Muahaha

```
1 | #include <signal.h>
2 | void divzero(int p){
3 |     while(true);}
4 | void segm(int p){
5 |     exit(0);}
6 | //in main
7 | signal(SIGFPE, divzero);
8 | signal(SIGSEGV, segm);
```

Mejorar velocidad

```
1 | ios::sync_with_stdio(false);
```

Mejorar velocidad 2

```
1 | //Solo para enteros positivos
2 | inline void Scanf(int& a){
```

```
3 |     char c = 0;
4 |     while(c<33) c = getc(stdin);
5 |     a = 0;
6 |     while(c>33) a = a*10 + c - '0', c = getc(stdin);
7 | }
```

Leer del teclado

```
1 | freopen("/dev/tty", "a", stdin);
```

File setup

```
1 | //tambien se pueden usar comas: {a, x, m, l}
2 | for i in {a..k}; do cp template.cpp $i.cpp && touch $i.in; done
```